Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 10, no. 1, pp. 69–94 (2006)

Finding Dominators in Practice

Loukas Georgiadis[1] *Robert E. Tarjan*[1,2] *Renato F. Werneck*[1]

[1]Department of Computer Science
Princeton University, Princeton, NJ 08544
http://www.cs.princeton.edu/
{lgeorgia,ret,rwerneck}@cs.princeton.edu

[2]Office of Strategy and Technology
Hewlett-Packard, Palo Alto, CA 94304
http://www.hp.com/

Abstract

The computation of dominators in a flowgraph has applications in several areas, including program optimization, circuit testing, and theoretical biology. Lengauer and Tarjan [30] proposed two versions of a fast algorithm for finding dominators and compared them experimentally with an iterative bit-vector algorithm. They concluded that both versions of their algorithm were much faster even on graphs of moderate size. Recently Cooper et al. [11] have proposed a new, simple, tree-based implementation of an iterative algorithm. Their experiments suggested that it was faster than the simple version of the Lengauer-Tarjan algorithm on graphs representing computer program control flows. Motivated by the work of Cooper et al., we present an experimental study comparing their algorithm (and some variants) with careful implementations of both versions of the Lengauer-Tarjan algorithm and with a new hybrid algorithm. Our results suggest that, although the performance of all the algorithms is similar, the most consistently fast are the simple Lengauer-Tarjan algorithm and the hybrid algorithm, and their advantage increases as the graph gets bigger or more complicated.

Article Type	Communicated by	Submitted	Revised
regular paper	Tomasz Radzik	September 2004	September 2005

Research at Princeton University partially supported by the Aladdin project, NSF Grant No. CCR-9626862.
A preliminary version of this paper was presented at the 12th Annual European Symposium on Algorithms (ESA 2004) [22].

1 Introduction

A flowgraph $G = (V, A, r)$ is a directed graph with $|V| = n$ vertices and $|A| = m$ arcs such that every vertex is reachable from a distinguished root vertex $r \in V$. A vertex w *dominates* a vertex v if every path from r to v includes w. Our goal is to find for each vertex v in V the set $Dom(v)$ of all vertices that dominate v. The dominance relation in G can be represented in compact form as a tree I, called the *dominator tree* of G, in which the dominators of a vertex v are its ancestors. Therefore, the output of a dominators computation can have size $\Theta(n)$.

Certain applications require computing the *postdominators* of G, defined as the dominators in the graph obtained from G by reversing all arc orientations. In this setting G is assumed to contain a sink vertex t reachable from all $v \in V$. Sometimes t is introduced as an artificial vertex of the graph. For example, in data-flow analysis t may represent a global exit point of a function.

The dominators problem occurs in several application areas, such as program optimization, code generation, circuit testing, and theoretical biology. Compilers make extensive use of dominance information during program analysis and optimization. Perhaps the best-known application of dominators is natural loop detection, which in turn enables a host of natural loop optimizations [32]. Structural analysis [36] also depends on dominance information. Postdominance information is used in calculating control dependencies in program dependence graphs [14]. Dominator trees are used in the computation of dominance frontiers [13], which are needed for efficiently computing program dependence graphs and static single-assignment forms. A dominator-based scheduling algorithm has also been proposed [39]. Apart from its applications in compilation, dominator analysis is also used in VLSI testing for identifying pairs of equivalent line faults in logic circuits [7]. In constraint programming, dominators have been used to implement generalized reachability constraints, which are helpful in the solution of the ordered disjoint-paths problem [35]. Another field where dominator analysis has been applied is theoretical biology; in [4, 5] dominators are used for the analysis of the extinction of species in trophic models (also called *foodwebs*).

The problem of finding dominators has been extensively studied. In 1972 Allen and Cocke showed that the dominance relation can be computed iteratively from a set of data-flow equations [3]. A direct implementation of this method has an $O(mn^2)$ worst-case time bound. Purdom and Moore [34] gave a straightforward algorithm with complexity $O(mn)$. It consists of performing a search in $G - v$ for each $v \in V$ (v dominates all the vertices that become unreachable from r). Improving on previous work by Tarjan [41], Lengauer and Tarjan [30] proposed an $O(m \log n)$-time algorithm and a more complicated $O(m\alpha(m, n))$-time version, where $\alpha(m, n)$ is an extremely slow-growing functional inverse of the Ackermann function [42]. Alstrup et al. [6] gave a linear-time algorithm for the random-access model of computation; a simpler algorithm was given by Buchsbaum et al. [9]. Georgiadis and Tarjan [20] gave a linear-time algorithm for the pointer machine computation model.

Experimental results for the dominators problem appear in [9, 11, 30]. In [30] Lengauer and Tarjan found the $O(m\alpha(m,n))$-time version of their algorithm (LT) to be faster than the simple $O(m \log n)$ version (SLT) even for small graphs. They also showed that the Purdom-Moore [34] algorithm is only competitive for graphs with fewer than 20 vertices, and that a bit-vector implementation of the iterative algorithm, by Aho and Ullman [2], is 2.5 times slower than LT for graphs with more than 100 vertices. Buchsbaum et al. [9] reported that their claimed linear-time algorithm has low constants, being only about 10% to 20% slower than their implementation of LT for graphs with more than 300 vertices. This algorithm was later shown to have the same time complexity as LT [20], and the corrected linear-time version is more complicated (see the Corrigendum of [9]). Cooper et al. [11] presented a clever tree-based space- and time-efficient implementation of the iterative algorithm, which they claimed to be 2.5 times faster than SLT. However, a more careful implementation of SLT later led to different results [10].

In this paper, we explore the effects of different initializations and processing orderings on the tree-based iterative algorithm. We also discuss implementation issues that make LT and SLT faster in practice and competitive with simpler algorithms even for small graphs. Furthermore, we describe a new algorithm that combines SLT (or LT) with the iterative algorithm and is very fast in practice. Finally, we present a thorough experimental analysis of the algorithms using real as well as artificial data. We have not included linear-time algorithms in our study; they are significantly more complex and thus unlikely to be faster than LT or SLT in practice.

2 Algorithms

In this section, we describe in more detail the algorithms we implemented.[1] Before doing so, we need to review some definitions and introduce additional notation. The *immediate dominator* of a vertex $v \neq r$, denoted by $idom(v)$, is the unique vertex $w \neq v$ that dominates v and is dominated by all vertices in $Dom(v) - v$ (the immediate dominator of r is undefined). The (immediate) *dominator tree* is a directed tree I rooted at r and formed by the arcs $\{(idom(v), v) \mid v \in V - r\}$. A vertex w dominates v if and only if w is an ancestor of v in I [1], so computing the immediate dominators is enough to determine all dominance information.

Given a directed graph $G = (V, A)$ we say that u is a *predecessor* of v (and that v is a *successor* of u) if $(u, v) \in A$. We denote the set of all predecessors of v by $pred(v)$ and the set of successors of v by $succ(v)$.

Throughout this paper the notation "$v \xrightarrow{*}_F u$" means that v is an ancestor of u in the forest F, and "$v \xrightarrow{+}_F u$" means that v is a proper ancestor of u in F. We use the same notation when we refer to the corresponding paths in F. Also, we omit the subscript when the context is clear. Given a tree T, we denote by

[1]Since most of the algorithms have been presented elsewhere, we do not show pseudocodes here. The interested reader will find them in [19].

T_v the subtree of T rooted at v, and by $p_T(v)$ the parent of v in T. If T is a spanning tree of a flowgraph $G = (V, A, r)$, an arc $a = (u, v) \in A$ is a *tree arc* (with respect to T) if $a \in T$, a *forward arc* if $u \xrightarrow{+}_T v$ and $(u, v) \notin T$, a *back arc* if $v \xrightarrow{+}_T u$, and a *cross arc* if u and v are unrelated in T. Given a forest F, we denote by $root_F(v)$ the root of the tree in F that contains v. Finally, for any subset $U \subseteq V$ and a tree T, $\mathrm{NCA}(T, U)$ denotes the nearest common ancestor of $U \cap T$ in T.

2.1 The Iterative Algorithm

Without loss of generality we can assume that the root r has no incoming arcs, since deleting such arcs has no effect on dominators. Then the sets $Dom(v)$ are the unique maximal solution to the following data-flow equations:

$$Dom'(v) = \Big(\bigcap_{u \in pred(v)} Dom'(u) \Big) \cup \{v\}, \ \forall\, v \in V. \tag{1}$$

As Allen and Cocke [3] showed, one can solve these equations iteratively by initializing $Dom'(r) = \{r\}$ and $Dom'(v) = V$ for $v \neq r$, and repeatedly applying the following step until it no longer applies:

> Find a vertex v such that (1) is false and replace $Dom'(v)$ by the expression on the right side of (1).

A simple way to perform this iteration is to cycle repeatedly through all the vertices of V until no $Dom'(v)$ changes. It is unnecessary to initialize all the sets $Dom'(v)$; it suffices to initialize $Dom'(r) = \{r\}$ and exclude uninitialized sets from the intersection in (1). In this case, an iterative step is applied to a vertex v only if a value has been computed for at least one $u \in pred(v)$. It is also possible to initialize the sets $Dom'(v)$ more accurately. Specifically, if S is any tree rooted at r (spanning or not, as long as it is a subgraph of G), we can initialize $Dom'(v)$ for $v \in S$ to be the set of ancestors of v in S, and leave $Dom'(v)$ for $v \notin S$ uninitialized.

Cooper et al. [11] improved the efficiency of this algorithm significantly by observing that we can represent all the sets $Dom'(v)$ by a single tree and perform an iterative step as an update of the tree. Specifically, we begin with any tree T rooted at r that is a subgraph of G and repeat the following step until it no longer applies:

> Find a vertex v such that
>
> $$pred(v) \cap T \neq \emptyset \text{ and } p_T(v) \neq \mathrm{NCA}(T, pred(v));$$
>
> replace $p_T(v)$ by $\mathrm{NCA}(T, pred(v))$.

The correspondence between this algorithm and the original algorithm is that for each vertex in T, $Dom'(v)$ is the set of ancestors of v in T. The intersection of $Dom'(u)$ and $Dom'(v)$ is the set of ancestors of $\mathrm{NCA}(T, \{u, v\})$ in T. Once the iteration stops, the current tree T is the dominator tree I. One can also perform the iteration arc-by-arc rather than vertex-by-vertex, replacing $p_T(v)$ by $\mathrm{NCA}(T, \{p_T(v), u\})$ for an arc (u, v) such that $u \in T$. The most straightforward implementation is to cycle repeatedly through the vertices (or arcs) until T does not change.

The number of iterations through the vertices (or arcs) depends on the order in which the vertices (or arcs) are processed. Kam and Ullman [28] show that certain data-flow equations, including (1), can be solved in at most $d(G, D) + 3$ iterations when the vertices are processed in reverse postorder with respect to a DFS tree D. Here $d(G, D)$ is the *loop connectedness of G with respect to D*, the largest number of back arcs found in any cycle-free path of G. When G is acyclic the dominator tree is built in one iteration. This is because reverse postorder is a topological sort of the vertices, so for any vertex v all vertices in $pred(v)$ are processed before v. The iterative algorithm will also converge in a single iteration if G is *reducible* [25], i.e., when the repeated application to G of the following operations

(i) delete a loop (v, v);

(ii) if (v, w) is the only arc entering $w \neq r$ delete w and replace each arc (w, x) with (v, x),

yields a single node. Equivalently, G is reducible if every loop has a single entry vertex from r. In a reducible flowgraph, v dominates u whenever (u, v) is a back arc [40]. Therefore, deletion of back arcs, which produces an acyclic graph, does not affect dominators.

The running time per iteration is dominated by the time spent on NCA computations. If these are performed naïvely (ascending the tree paths until they meet), then a single iteration takes $O(mn)$ time. Because there may be up to $\Theta(n)$ iterations, the running time is $O(mn^2)$. The iterative algorithm runs much faster in practice, however. Typically $d(G, D) \leq 3$ [29], and it is reasonable to expect that few NCA calculations will require $\Theta(n)$ time. If T is represented as a dynamic tree [37], the worst-case bound per iteration is reduced to $O(m \log n)$, but the implementation becomes much more complicated and unlikely to be practical.

Initializations and vertex orderings. Our base implementation of the iterative algorithm (IDFS) starts with $T \leftarrow \{r\}$ and processes the vertices in reverse postorder with respect to a DFS tree; this algorithm is the one proposed by Cooper et al. [11]. It requires a preprocessing phase that performs a DFS on the graph and assigns a postorder number to each vertex. We do not initialize T as a DFS tree because this is bad both in theory and in practice: it causes the back arcs to be processed in the first iteration, even though they contribute nothing to the NCAs in this case.

Intuitively, a much better initial approximation of the dominator tree is a BFS tree. We implemented a variant of the iterative algorithm (which we call

IBFS) that starts with such a tree and processes the vertices in BFS order. As Section 4 shows, this method is often (but not always) faster than IDFS.

We note that there is an ordering σ of the arcs that is optimal with respect to the number of iterations that are needed for convergence. This is stated in the following lemma.

Lemma 1 *There exists an ordering σ of the arcs of G such that if the iterative algorithm processes the arcs according to σ, then it will construct the dominator tree of G in a single iteration.*

Proof: We will use ancestor-dominance spanning trees as defined in [21]. There it is shown that G has two spanning trees T_1 and T_2 such that, for any v, the paths $r \overset{*}{\rightarrow}_{T_1} v$ and $r \overset{*}{\rightarrow}_{T_2} v$ intersect only at the vertices of $Dom(v)$. We construct σ by catenating a list σ_1 of the arcs of T_1 with a list σ_2 of the arcs of T_2 that do not appear in T_1. The arcs in σ_i are sorted lexicographically in ascending order with respect to a preorder numbering of T_i. The iterative algorithm starts with $T \leftarrow \{r\}$. After processing the arcs of σ_1 we will have $T = T_1$. We show by induction that after $(p_{T_2}(v), v)$ is processed in σ_2 we will have $p_T(v) = idom(v)$. This is immediate for any child of r in T_2. Suppose now that $u = p_{T_2}(v) \neq r$. Since $(p_{T_2}(u), u)$ has been processed before (u, v), we have by the induction hypothesis that $x = \text{NCA}(T, \{u, v\})$ is a dominator of u. Thus, by the ancestor-dominance property, x is an ancestor of u in both T_1 and T_2. Since u is on $r \overset{*}{\rightarrow}_{T_2} v$, x is also an ancestor of v in T_2. Note that when a vertex w moves to a new parent in T, it ascends the path $r \overset{*}{\rightarrow}_{T} w$. So, the set of ancestors of w in T is always a subset of the set of its ancestors in T_1. This implies that x is also an ancestor of v in T_1, and by the ancestor-dominance property, x dominates v. Now notice that $idom(v)$ dominates u. If this were not the case there would be a path P from r to u that avoids $idom(v)$. Then P followed by (u, v) would form a path from r to v that avoids $idom(v)$, a contradiction. Therefore, $idom(v)$ is both on $r \overset{*}{\rightarrow}_{T} v$ and on $r \overset{*}{\rightarrow}_{T} u$, and using the fact that $x \in Dom(v)$ we conclude that $x = idom(v)$. □

This generalizes the result of Hecht and Ullman [25], which considered the restricted class of reducible graphs. An algorithm given in [21] constructs ancestor-dominance spanning trees by running a modified version of the Lengauer-Tarjan algorithm, and therefore takes $O(m\alpha(m, n))$ time (or linear time using a more complicated algorithm [8, 19]). An open question, of both theoretical and practical interest, is whether there is a simple linear-time construction of such an ordering.

Another question that arises immediately is whether any graph has a *fixed* ordering of its vertices that guarantees convergence of the iterative algorithm in a constant number of iterations. The answer is negative. Figure 1 shows linearvit(k), a graph family that requires $\Theta(k) = \Theta(n)$ iterations. A similar graph was used in [12] in order to show that by processing the vertices in reverse postorder, $\Theta(n)$ iterations are necessary. The next lemma shows that in fact this graph requires $\Theta(n)$ iterations for any fixed ordering of the vertices. Note that,

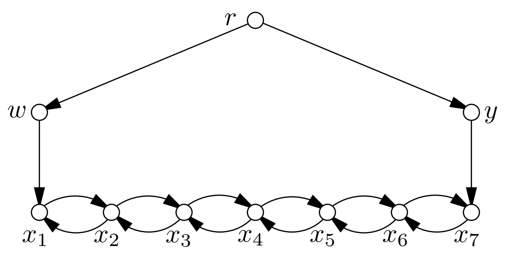

Figure 1: Graph family linearvit(k). In this instance $k = 7$. The iterative algorithm needs $\Theta(k)$ iterations to converge when we initialize $T \leftarrow \{r\}$, and the vertices are processed in any fixed order in each iteration.

if we allow the vertex ordering to be different in each iteration, then the proof of Lemma 1 trivially implies that two iterations suffice to build the dominator tree.

Lemma 2 *Any iterative algorithm that processes the vertices in a fixed order requires $\Theta(k)$ iterations to build the dominator tree of linearvit(k).*

Proof: Consider the time when the first vertex x_i has its parent set to r. This cannot happen until a tree has been built that is a subgraph of the original graph and contains all vertices but x_i; after this happens, the lone vertex x_i that is not in the tree can have its parent set to r. Suppose without loss of generality that $i \geq k/2$. After x_i is processed, the current tree T consists of a path from r to x_{i-1} through w and $x_1, x_2, \ldots, x_{i-2}$, a path from r to x_{i+1} through y and $x_k, x_{k-1}, \ldots, x_{i+2}$ and an arc from r to x_i. Let p be the number of pairs x_j, x_{j+1} such that x_{j+1} follows x_j in the vertex order and $1 \leq j < i-1$. The number of passes through the vertices required to build the tree T is at least $i-1-p$. Once x_i is processed, the vertices $x_{i-1}, x_{i-2}, \ldots, x_1$, in this order, will eventually become children of r. The number of passes required for this is at least p, because if a pair x_j, x_{j+1} is such that x_{j+1} follows x_j in the vertex order, then x_j must have its parent set to r in a pass after the pass that sets the parent of x_{j+1} to r. This means that there must be at least $i-1-p+p = i-1 \geq k/2-1$ passes. This bound is tight to within the additive constant. □

Marking. It is reasonable to expect that not all the $Dom'(v)$ sets will be updated in each iteration. We could obtain significant savings if we could locate and mark the vertices that need to be processed in the next iteration. However, we cannot locate these vertices efficiently, since assigning a new parent in T to a vertex v may require marking all the successors in G of each vertex in T_v. Indeed, as it turned out in our experiments, this marking scheme added a significant overhead to the iterative algorithm, which was much slower on most graphs (the exceptions were a few artificial graphs). Hence we did not include these experiments in Section 4.

2.2 The Lengauer-Tarjan Algorithm

The Lengauer-Tarjan algorithm starts with a depth-first search on G from r and assigns preorder numbers to the vertices. The resulting DFS tree D is represented by an array *parent*. For simplicity, we refer to the vertices of G by their preorder number, so $v < u$ means that v has a lower preorder number than u. The algorithm is based on the concept of *semidominators*, which give an initial approximation to the immediate dominators. We call a path $P = (u = v_0, v_1, \ldots, v_{k-1}, v_k = v)$ in G a *semidominator path* if $v_i > v$ for $1 \le i \le k-1$. The semidominator of v is defined as

$$sdom(v) = \min\{u \mid \text{there is a semidominator path from } u \text{ to } v\}.$$

Any vertex $v \neq r$ is related to $idom(v)$ and $sdom(v)$ in the following way

$$idom(v) \xrightarrow{*} sdom(v) \xrightarrow{+} v. \tag{2}$$

Semidominators and immediate dominators are computed by finding minimum *sdom* values on paths of D. As shown in [30], for any vertex $w \neq r$,

$$sdom(w) = \min\{s_w(v) \mid v \in pred(w)\},$$

where the function $s_w : pred(w) \mapsto V$ is defined as

$$s_w(v) = \begin{cases} v, & v \le w \\ \min\{sdom(u) \mid \mathrm{NCA}(D, \{v, w\}) \xrightarrow{+} u \xrightarrow{*} v\}, & v > w \end{cases}.$$

The immediate dominator can be found similarly, by evaluating the function $e : V - r \mapsto V$, defined by

$$e(w) = \arg\min\{sdom(u) \mid sdom(w) \xrightarrow{+} u \xrightarrow{*} w\}.$$

For any $w \neq r$, $e(w)$ is a *relative dominator* of w, i.e., it satisfies $idom(e(w)) = idom(w)$. Furthermore, if $sdom(e(w)) = sdom(w)$ then $idom(w) = sdom(w)$.

The vertices are processed in reverse preorder to ensure that all the necessary information is available when needed. The core of the computation is performed by a *link-eval* data structure, introduced by Tarjan [43]. Given a tree T on V and real values $value(v)$ for each v in V, the link-eval data structure maintains a forest F that is a subgraph of T subject to the following operations:

$link(v, x)$: Assign $value(v) \leftarrow x$ and add arc $(p_T(v), v)$ to F. This links the tree rooted at v in F to the tree rooted at $p_T(v)$ in F.

$eval(v)$: If $v = root_F(v)$, return v. Otherwise, return any vertex of minimum value among the vertices u that satisfy $root_F(v) \xrightarrow{+}_F u \xrightarrow{*}_F v$.

Initially, every vertex v in V is a singleton in F. We say that v is *linked* if $\mathit{link}(v, \cdot)$ has been performed. In order to enhance the performance of the *eval* operation, the link-eval data structure employs *path compression* and other techniques [43]. Instead of dealing with F directly, it actually maintains a *virtual forest*, denoted by VF, and ensures that the *eval* operation applied on VF returns the same value as if it were applied on F.

In the Lengauer-Tarjan algorithm, $T = D$ and, for all $v \in V$, $\mathit{value}(v)$ equals v initially and $\mathit{sdom}(v)$ after v is processed for the first time. Every vertex w is processed three times. The first time w is processed, $\mathit{sdom}(w)$ is computed by executing $\mathit{eval}(u)$ for each u in $\mathit{pred}(w)$, thus computing $s_w(u)$. Then w is inserted into a bucket associated with vertex $\mathit{sdom}(w)$ and $\mathit{link}(w, \mathit{sdom}(w))$ is performed. The algorithm processes w again after $\mathit{sdom}(v)$ has been computed, where v satisfies $\mathit{parent}[v] = \mathit{sdom}(w)$ and $v \xrightarrow{*} w$; at this time it performs the operation $\mathit{eval}(w)$, thus computing $e(w)$. Finally, immediate dominators are derived from relative dominators in a preorder pass.

With a simple implementation of the link-eval data structure, using only path compression, the Lengauer-Tarjan algorithm runs in $O(m \log_{(2+m/n)} n)$ time [44]. With a more elaborate linking strategy that ensures that VF is balanced, the algorithm runs in $O(m\alpha(m, n))$ time [43]. We refer to these two versions as SLT and LT, respectively.

Implementation issues. Lengauer and Tarjan process $\mathit{bucket}[\mathit{parent}[w]]$ at the end of the iteration that deals with w; hence the same bucket may be processed several times. A better alternative is to process $\mathit{bucket}[w]$ in the beginning of the iteration that deals with w; each bucket is now processed exactly once, so it need not be emptied explicitly.

We observe that buckets have very specific properties: (1) every vertex is inserted into at most one bucket; (2) there is exactly one bucket associated with each vertex; (3) vertex i can only be inserted into some bucket after bucket i itself is processed. Properties (1) and (2) ensure that buckets can be implemented with two n-sized arrays, *first* and *next*: $\mathit{first}[i]$ represents the first element in bucket i, and $\mathit{next}[v]$ is the element that succeeds v in the bucket it belongs to. Property (3) ensures that these two arrays can actually be combined into a single array *bucket*.

Another measure that is relevant in practice is to avoid unnecessary bucket insertions: a vertex w for which $\mathit{parent}[w] = \mathit{sdom}(w)$ is not inserted into any bucket because we already know that $\mathit{idom}(w) = \mathit{parent}[w]$. Also, we note that the last bucket to be processed is the one associated with the root r. For all vertices v in this bucket, we just need to set $\mathit{idom}(v) \leftarrow r$; there is no need to call *eval* for them.

2.3 The SEMI-NCA Algorithm

In this section, we introduce SEMI-NCA, a new hybrid algorithm for computing dominators that works in two phases:

(a) Compute $sdom(v)$ for all $v \neq r$, as done by Lengauer-Tarjan.

(b) Build I incrementally as follows: Process the vertices in preorder. For each vertex w, ascend the path $r \xrightarrow{*}_I p_D(w)$ (where D is the DFS tree used in phase (a)) until reaching the deepest vertex x such that $x \leq sdom(w)$, and set x to be the parent of w in I.

The correctness of this algorithm is based on the following result:

Lemma 3 *For any vertex $w \neq r$, $idom(w)$ is the nearest common ancestor in I of $sdom(w)$ and $p_D(w)$, i.e.,*

$$idom(w) = \mathrm{NCA}(I, \{p_D(w), sdom(w)\}).$$

Proof: By relation (2), we have that $idom(w) \xrightarrow{*} sdom(w) \xrightarrow{*} p_D(w)$. Obviously, if $p_D(w) = idom(w)$ then $idom(w) = sdom(w)$ and we are done. Now suppose $p_D(w) \neq idom(w)$. Then also $p_D(w) \neq sdom(w)$. First we observe that any vertex u that satisfies $idom(w) \xrightarrow{+} u \xrightarrow{*} w$ is dominated by $idom(w)$; if not then there would be a path from r to u that avoids $idom(w)$, which catenated with $u \xrightarrow{*} w$ forms a path from r to w that avoids $idom(w)$, a contradiction. Hence, both $sdom(w)$ and $p_D(w)$ are dominated by $idom(w)$.

If $idom(w) = sdom(w)$, then clearly $idom(w)$ is the nearest common ancestor of $sdom(w)$ and $p_D(w)$ in I. Now suppose $idom(w) \neq sdom(w)$. Let v be any vertex such that $idom(w) \xrightarrow{+} v \xrightarrow{*} sdom(w)$. Then there is a path P from $idom(w)$ to w that avoids v. Let z be the first vertex on P that satisfies $v \xrightarrow{+} z \xrightarrow{*} p_D(w)$. This vertex must exist, since otherwise $sdom(w) < v$, which contradicts the choice of v. Therefore v cannot be a dominator of $p_D(w)$. Since $idom(w)$ dominates both $p_D(w)$ and $sdom(w)$, we conclude that $idom(w)$ is nearest common ancestor of $sdom(w)$ and $p_D(w)$ in I. □

The above lemma implies that $x = idom(w)$. Although we gave a direct proof of this fact we note that it also follows by applying the iterative algorithm to the graph $G' = (V, A', r)$, where A' consists of the arcs $(p_D(w), w)$ and $(sdom(w), w)$, for all $w \in V - r$. Clearly the semidominator of any vertex is the same in both G and G'. Hence the Lengauer-Tarjan algorithm implies that the dominators are also the same. Finally, since G' is acyclic the iterative algorithm IDFS builds the dominator tree in one iteration.

If we perform the search for $idom(w)$ naïvely, by visiting all the vertices on the path $idom(w) \xrightarrow{*}_I p_D(w)$, the second phase runs in $O(n^2)$ worst-case time. However, we expect it to be much faster in practice, since our empirical results indicate that $sdom(v)$ is usually a good approximation to $idom(v)$.

SEMI-NCA is simpler than the Lengauer-Tarjan algorithm in three ways. First, *eval* can return the minimum value itself rather than a vertex that achieves that value. This eliminates one array and one level of indirect addressing. Second, buckets are no longer necessary because the vertices are processed in preorder in the second phase. Finally, there is one fewer pass over the vertices, since there is no need to compute immediate dominators from relative dominators. Therefore, the algorithm aims to combine the efficiency of SLT with the simplicity of the iterative algorithm.

With the simple implementation of *link* and *eval* (which is faster in practice), this method (which we call SNCA) runs in $O(n^2)$ worst-case time. We note that Gabow [16] has given a rather complex procedure that computes NCAs in total linear time on a tree that grows through the addition of leaves. Therefore, the time for the second phase of SEMI-NCA can be reduced to $O(n)$, but this is unlikely to be practical. In fact, Gabow's result (together with Lemma 3) would yield a linear-time algorithm for computing dominators, if one could compute semidominators in linear time.

3 Worst-Case Behavior

This section describes families of graphs that elicit the worst-case behavior of the algorithms we implemented. In particular, they show that neither IBFS nor IDFS dominate the other: there are instances on which IBFS is asymptotically faster than IDFS, and vice versa. The worst-case graphs also confirm that the time bounds we have presented for SNCA, IBFS, and IDFS are tight. Although such graphs are unlikely to appear in practice, it is important to study them because similar patterns may occur in real-world instances. Also, as discussed by Gal et al. [17, 18], in an extreme situation a malicious user could exploit the worst-case behavior of a compiler to launch a denial-of-service attack.

Figure 2 shows graph families that favor particular methods against the others. For each family, we define a parameter k that controls the size of its members. We denote by $G_k = (V_k, A_k)$ the member that corresponds to a particular value of k.

Iterative. Family itworst(k) contains worst-case inputs for the iterative methods. The set of vertices V_k is defined as $\{r\} \cup \{w_i, x_i, y_i, z_i \mid 1 \leq i \leq k\}$. The set of arcs A_k is the union of

$$\{(r, w_1), (r, x_1), (r, z_k)\}, \{(w_i, w_{i+1}), (x_i, x_{i+1}), (y_i, y_{i+1}), (z_i, z_{i+1}) \mid 1 \leq i < k\},$$

$$\{(z_i, z_{i-1}) \mid 1 < i \leq k\}, \{(x_k, y_1), (y_k, z_1)\}, \text{ and } \{(y_i, w_j) \mid 1 \leq i, j \leq k\}.$$

We have $|V_k| = 4k + 1$ and $|A_k| = k^2 + 5k$. Because of the chain of k back arcs (z_i, z_{i-1}), the iterative methods need $\Theta(k)$ iterations to converge. Each iteration requires $\Theta(k^3)$ operations to process the k^2 arcs (y_i, w_j), so the total running time is $\Theta(k^4)$.

Note however that only the dominators of the z_i's change after the first iteration. This fact can be detected by marking the vertices that need to be processed in each iteration, thus processing only the z_i's after the second iteration, and finishing in $\Theta(k^3)$ total time. If we added the arc (z_1, y_k), then all the vertices would have to be marked in each iteration and the total running time would remain $\Theta(k^4)$. As already mentioned, this marking scheme did not work well in our experiments, so we did not include results for it in Section 4.

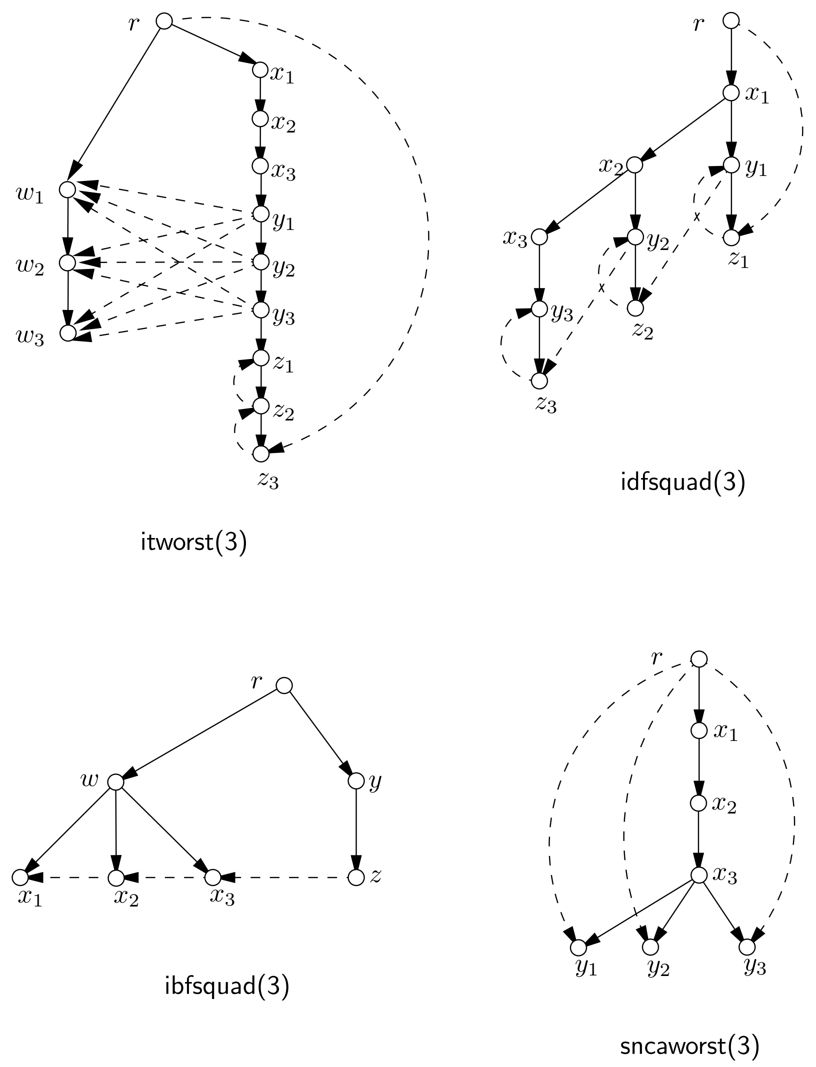

Figure 2: Worst-case families for $k = 3$. Subscripts take values in $\{1, \ldots, k\}$. The vertices with no subscript are fixed for each member of a family. The DFS tree used by IDFS, SLT, LT and SNCA is shown with solid arcs; the arcs outside the DFS tree are dashed. Note that other initial DFS trees (though not all) would also cause bad behavior.

IDFS. Family idfsquad(k) favors IBFS over IDFS. V_k contains the vertices in $\{r\}$ and $\{x_i, y_i, z_i \mid 1 \leq i \leq k\}$. A_k is the union of

$$\{(r, x_1), (r, z_1)\}, \{(x_i, x_{i+1}), (y_i, z_{i+1}) \mid 1 \leq i < k\} \text{ and}$$

$$\{(x_i, y_i), (y_i, z_i), (z_i, y_i) \mid 1 \leq i \leq k\}.$$

We have $|V_k| = 3k+1$ and $|A_k| = 5k$. By processing the vertices in reverse postorder, IDFS requires $k+1$ iterations to propagate the correct dominator values from z_1 to y_k, and the total running time is $\Theta(k^2)$. On the other hand, IBFS processes the vertices in the correct order. Therefore, it constructs the dominator tree in one iteration and runs in linear time (as do the semidominator-based methods).

IBFS. Family ibfsquad(k) favors IDFS over IBFS. Here V_k is the union of $\{r, w, y, z\}$ and $\{x_i, \mid 1 \leq i \leq k\}$. A_k contains the arcs (r, w), (r, y), (y, z), and (z, x_k), alongside with the sets

$$\{(w, x_i) \mid 1 \leq i \leq k\} \text{ and } \{(x_i, x_{i-1}) \mid 1 < i \leq k\}.$$

Then $|V_k| = k+4$ and $|A_k| = 2k+3$. Processing the vertices in BFS order takes k iterations to reach the fixed point. On the other hand, one iteration with cost $O(k)$ suffices if we order the vertices in reverse postorder, since the graph is acyclic. The semidominator-based methods also run in linear time.

Simple Lengauer-Tarjan. Family sltworst(k) causes worst-case behavior of SLT [15, 44]. For any particular k this is a graph with k vertices ($r = x_1, \ldots, x_k$) and $2k-2$ arcs that causes path compression without balancing to take $\Theta(k \log k)$ time. The graph contains the arcs (x_i, x_{i+1}), $1 \leq i < k$, and $k-1$ arcs (x_i, x_j) where $j < i$, with the property that, after x_j is linked, x_i is a vertex with maximum depth in the tree rooted at x_j of the virtual forest VF. Note that $idom(x_i) = x_{i-1}$ for every $i > 1$. For this reason, the iterative methods need only one iteration to build the dominator tree. However, they still run in quadratic time because they process the same paths repeatedly. It is unclear whether there exists a graph family on which the iterative algorithm runs asymptotically faster than SLT.

SEMI-NCA. Family sncaworst(k) causes the worst-case behavior of SNCA. The set of vertices V_k consists of r, x_i and y_i for $1 \leq i \leq k$. The set of arcs A_k is the union of

$$\{(r, x_1)\}, \{(x_i, x_{i+1}) \mid 1 \leq i < k\} \text{ and } \{(r, y_i), (x_k, y_i) \mid 1 \leq i \leq k\}.$$

We have $|V_k| = 2k+1$ and $|A_k| = 3k$. Note that $sdom(y_i) = r$ and x_k is the parent of every y_i, which forces SNCA to ascend the path from x_k to r for every y_i. As a result, the algorithm runs in $\Theta(k^2)$ total time. The same bound holds for the iterative methods as well (despite the fact that the graph is reducible), as they also have to traverse the same long path repeatedly. On the other hand, the Lengauer-Tarjan algorithm can handle this situation efficiently because of path compression.

Notice that if we added any arc (y_i, x_k), then we would have $sdom(x_k) = r$ and SNCA would run in linear time. Also BFS would set y_i to be the parent of x_k and r to be the parent of y_i, which implies that IBFS would also run in $O(k)$ time. However, IDFS would still need quadratic time.

4 Empirical Analysis

Based on worst-case bounds only, the sophisticated version of the Lengauer-Tarjan algorithm is the method of choice among those studied here. In practice, however, "sophisticated" algorithms tend to be harder to code and to have higher constants, so other alternatives might be preferable. The experiments reported in this section shed some light on this issue.

Implementation and experimental setup. We implemented all algorithms in C++. They take as input the graph and its root, and return an n-element array representing immediate dominators. Vertices are assumed to be integers from 1 to n. Within reason, we made all implementations as efficient and uniform as we could (of course, there might still be room for further improvement, especially for the more sophisticated algorithms). The source code is available from the authors upon request.

The code was compiled using `g++` v. 3.3.1 with full optimization (flag `-O4`). All tests were conducted on a Pentium IV with 256 MB of RAM and 256 kB of cache running Mandrake Linux 9.2 at 1.7 GHz. We report CPU times measured with the `getrusage` function. Since the precision of `getrusage` is only 1/60 second, we ran each algorithm repeatedly for at least one second; individual times were obtained by dividing the total time by the number of runs. Note that this strategy artificially reduces the number of cache misses for all algorithms, since the graphs are usually small. To minimize fluctuations due to external factors, we used the machine exclusively for tests, took each measurement three times, and picked the best. Running times do not include creating the graph (which uses an array-based representation) or creating predecessor lists from successor lists (both required by all algorithms). However, times do include allocating and deallocating the arrays used by each method.

Instances. We used control-flow graphs produced by the SUIF compiler [26] from benchmarks in the SPEC'95 suite [38] and previously tested by Buchsbaum et al. [9] in the context of dominator analysis. We also used control-flow graphs created by the IMPACT compiler [27] from six programs in the SPEC 2000 suite. The instances were divided into *series*, each corresponding to a single benchmark. Series were further grouped into three classes, SUIF-FP, SUIF-INT, and IMPACT. We also considered two variants of IMPACT: class IMPACTP contains the reverse graphs and is meant to test how effectively the algorithms compute postdominators; IMPACTS contains the same instances as IMPACT, with parallel arcs removed. (These arcs appear in optimizing compilers due to superblock formation, and are produced much more often by IMPACT than by SUIF.) We also ran the algorithms on graphs representing circuits from VLSI-testing applications [7] obtained from the ISCAS'89 suite [45] (all 50 graphs were considered a single class), and on graphs representing foodwebs used in [4, 5] (all 21 graphs were considered a single class).

Finally, we tested eight instances that do not occur in any particular application related to dominators. Five are instances from the worst-case families described in Section 3, and the other three are large graphs representing speech recognition finite state automata (originally used to test dominator algorithms by Buchsbaum et al. [9]).

Test results. We start with the following experiment: read an entire series into memory and compute dominators for each graph in sequence, measuring the total running time. This simulates the behavior of a compiler, which must process several graphs (which typically represent functions) to produce a single executable.

For each series, Table 1 shows the total number of graphs (g) and the average number of vertices and arcs (n and m). As a reference, we report the average time (in microseconds) of a simple breadth-first search (BFS) on each graph. Since all the algorithms considered here start by performing a graph search, BFS acts as a lower bound on the actual running time and therefore can be used as a baseline for comparison.[2] Times for computing dominators are given as multiples of BFS. Using a baseline method is a standard technique for evaluating algorithms whose running time is close to linear in practice [23, 24, 31, 33].

In absolute terms, all algorithms are reasonably fast: none is slower than BFS by a factor of more than six on compiler-generated graphs. The worst relative time observed was slightly below eight, for FOODWEB. Furthermore, despite their different worst-case complexities, all methods have remarkably similar behavior in practice. In no series was an algorithm twice as fast (or slow) as any other. Differences do exist, of course. LT is consistently slower than SLT, which can be explained by the complex nature of LT and the relatively small size of the instances tested. The iterative methods are usually faster than LT, but often slower than SLT. Both variants (IDFS and IBFS) usually have very similar behavior, although occasionally one method is significantly faster than the other (series 145.fppp and 256.bzip2 are good examples). Almost always within a factor of four of BFS (with FOODWEB as the only exception), SNCA and SLT are the most consistently fast methods in the set.

By measuring the total (or average) time per series, the results are naturally biased towards large graphs. For a more complete view, we also computed running times for individual instances, and normalized them with respect to BFS. In other words, for each individual instance we calculated the ratio between the running times of the dominator algorithm and of BFS (the result is the

[2]Another natural baseline algorithm is DFS; the results are similar.

Table 1: Complete series: number of graphs (g), average number of vertices (n) and arcs (m), and average time per graph (in microseconds for BFS, and relative to BFS for all dominator algorithms). The best result in each row is marked in bold.

INSTANCE		DIMENSIONS			BFS	RELATIVE TOTAL TIMES				
CLASS	SERIES	g	n	m	TIME	IBFS	IDFS	LT	SLT	SNCA
CIRCUITS	circuits	50	3228.8	5027.2	224.72	6.42	5.44	4.87	3.87	**3.49**
FOODWEB	foodweb	21	78.8	741.0	5.10	7.73	7.88	6.72	4.59	**4.50**
IMPACT	181.mcf	26	26.5	90.3	1.31	4.47	5.00	5.34	3.56	**3.53**
	197.parser	324	16.8	55.7	1.17	3.69	4.63	4.53	3.26	**3.19**
	254.gap	854	25.3	56.2	1.86	2.89	3.26	3.82	2.77	**2.63**
	255.vortex	923	15.1	35.8	1.25	2.62	2.95	3.44	2.50	**2.34**
	256.bzip2	74	22.8	70.3	1.24	3.93	5.13	5.03	3.45	**3.32**
	300.twolf	191	39.5	115.6	2.38	4.33	5.00	5.00	3.57	**3.34**
IMPACTP	181.mcf	26	26.5	90.3	1.34	4.40	5.16	5.03	3.56	**3.32**
	197.parser	324	16.8	55.7	1.20	3.44	4.30	4.13	3.00	**2.93**
	254.gap	854	25.3	56.2	1.77	3.57	3.60	3.77	2.81	**2.78**
	255.vortex	923	15.1	35.8	1.25	2.80	3.12	3.31	2.48	**2.45**
	256.bzip2	74	22.8	70.3	1.27	3.80	5.16	4.70	3.34	**3.16**
	300.twolf	191	39.5	115.6	2.29	4.75	5.43	4.94	3.54	**3.43**
IMPACTS	181.mcf	26	26.5	72.4	1.30	3.88	4.53	4.93	3.31	**3.21**
	197.parser	324	16.8	42.1	1.09	3.42	4.11	4.30	3.23	**3.04**
	254.gap	854	25.3	48.8	1.79	2.76	3.08	3.85	2.79	**2.65**
	255.vortex	923	15.1	27.1	1.15	2.43	2.65	3.41	2.48	**2.36**
	256.bzip2	74	22.8	53.9	1.16	3.56	4.40	4.95	3.40	**3.29**
	300.twolf	191	39.5	96.5	2.16	4.20	4.90	5.14	3.72	**3.49**
SUIF-FP	101.tomcatv	1	143.0	192.0	4.19	3.87	3.55	5.59	3.63	**3.48**
	102.swim	7	26.6	34.4	0.98	3.07	2.97	4.43	3.10	**2.84**
	103.su2cor	37	32.3	42.7	1.31	**2.95**	2.99	4.45	2.99	**2.95**
	104.hydro2d	43	35.3	47.0	1.42	3.02	2.88	4.43	2.95	**2.79**
	107.mgrid	13	27.2	35.4	1.08	2.99	2.79	4.18	2.89	**2.76**
	110.applu	17	62.2	82.8	2.13	3.46	3.23	4.93	3.34	**3.20**
	125.turb3d	24	54.0	73.5	1.54	3.66	3.59	5.86	3.70	**3.25**
	145.fpppp	37	20.3	26.4	0.81	3.49	3.18	4.77	3.18	**3.10**
	146.wave5	110	37.4	50.7	1.43	3.19	3.31	4.90	3.27	**3.10**
SUIF-INT	009.go	372	36.6	52.5	1.62	3.29	3.51	4.93	**3.27**	3.29
	124.m88ksim	256	27.0	38.7	1.13	3.32	3.52	5.11	**3.27**	3.29
	126.gcc	2013	48.3	69.8	2.34	3.03	3.12	4.46	**2.94**	**2.94**
	129.compress	24	12.6	16.7	0.62	2.70	3.15	3.83	2.74	**2.66**
	130.li	357	9.8	12.8	0.53	**2.44**	2.74	3.81	2.73	2.66
	132.ijpeg	524	14.8	20.1	0.81	**2.66**	3.12	4.35	2.92	3.00
	134.perl	215	66.3	98.2	2.69	3.72	3.97	5.37	3.54	**3.52**
	147.vortex	923	23.7	34.9	1.36	2.63	2.72	3.73	2.59	**2.48**

Table 2: Times relative to BFS: geometric mean and geometric standard deviation. The lowest mean in each row is marked in bold.

	IBFS		IDFS		LT		SLT		SNCA	
CLASS	MEAN	DEV	MEAN	DEV	MEAN	DEV	MEAN	DEV	MEAN	DEV
CIRCUITS	6.12	1.42	5.97	1.23	6.37	1.16	4.60	1.15	**4.26**	1.13
FOODWEB	7.01	1.31	7.11	1.32	6.52	1.13	4.22	1.19	**4.14**	1.19
SUIF-FP	**2.56**	1.56	2.71	1.40	4.03	1.42	2.88	1.31	2.78	1.26
SUIF-INT	**2.51**	1.56	2.81	1.42	3.98	1.43	2.80	1.31	2.71	1.29
IMPACT	**2.36**	1.70	2.87	1.60	3.80	1.40	2.76	1.32	2.61	1.30
IMPACTP	**2.42**	1.78	2.83	1.59	3.60	1.38	2.59	1.31	2.55	1.29
IMPACTS	**2.25**	1.64	2.64	1.56	3.71	1.41	2.70	1.33	2.55	1.32

relative time of the algorithm). For each class, Table 2 shows the geometric mean and the geometric standard deviation of the relative times. Now that each graph is given equal weight, the aggregate measures for iterative methods (IBFS and IDFS) are somewhat better than before, particularly for IMPACT instances. This, together with the fact that their deviations are higher, suggests that iterative methods are faster than semidominator-based methods for small instances, but slower when size increases.

The plot in Figure 3 confirms this for the IMPACT class. Each point represents the mean relative running times for all graphs with the same value of $\lceil \log_2(n+m) \rceil$. Iterative methods clearly have a much stronger dependence on size than other algorithms. Almost as fast as a single BFS for very small instances, they become the slowest alternatives as size increases. The relative performance of the other methods is the same regardless of size: SNCA is slightly faster than SLT, and both are significantly faster than LT. A similar behavior was observed for IMPACTS and IMPACTP.

For SUIF, which contains graphs that are somewhat simpler, iterative methods remained competitive even for larger sizes. This is shown in Figure 4 for SUIF-INT (the results for SUIF-FP are similar). Note that SLT and SNCA still have better performance as the graph size increases, but now they are closely followed by the iterative methods. All algorithms tend to "level-off" with respect to BFS as the size increases, which suggests an almost-linear behavior for this particular class.

Figure 5 presents the corresponding results for class CIRCUIT. For the range of sizes shown (note that the graphs are bigger than in the other classes), the average performance of each algorithm (relative to BFS) does not have a strong correlation with graph size.

Finally, Figure 6 contains results for the FOODWEB class. On these graphs, which are significantly denser than the others, LT starts to outperform the iterative methods much sooner.

The results for IMPACT and IMPACTS shown in Tables 1 and 2 indicate that the iterative methods benefit the most by the absence of parallel arcs.

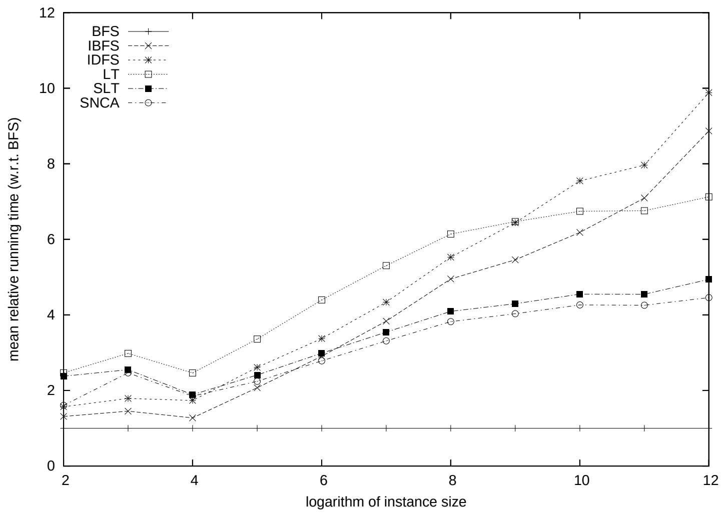

Figure 3: Times for IMPACT instances within each size. Each point represents the mean relative running time (w.r.t. BFS) for all instances with the same value of $\lceil \log_2(n+m) \rceil$.

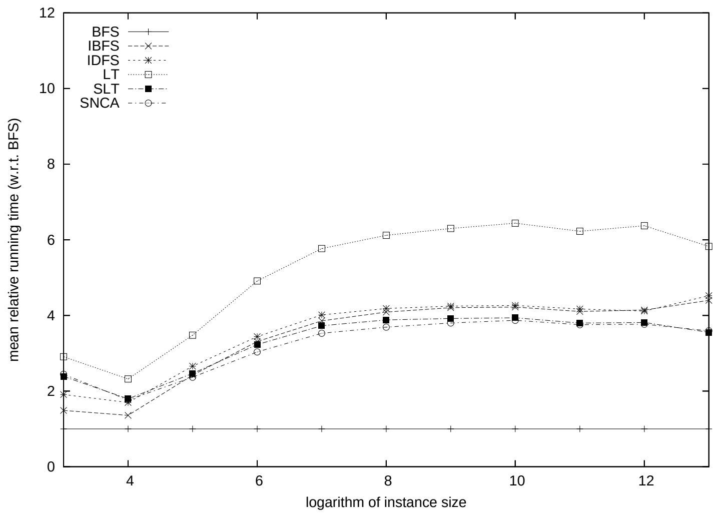

Figure 4: Times for SUIF-INT instances within each size. Each point is the mean relative running time (w.r.t. BFS) for instances with the same value of $\lceil \log_2(n+m) \rceil$.

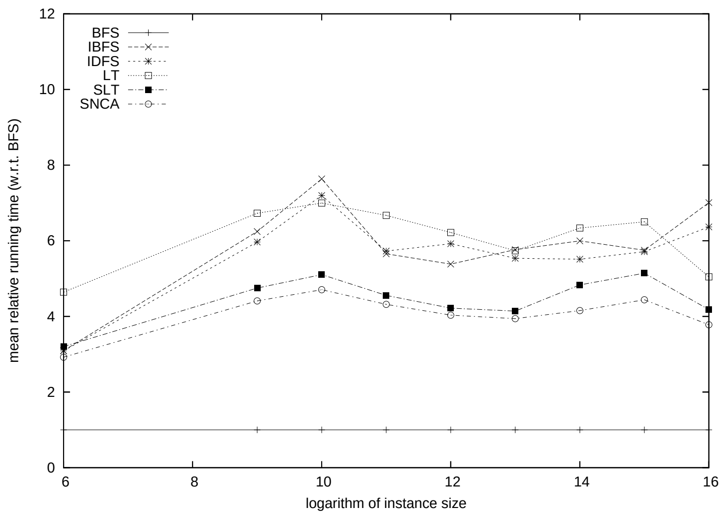

Figure 5: Times for CIRCUIT instances within each size. Each point is the mean relative running time (w.r.t. BFS) for instances with the same value of $\lceil \log_2(n+m) \rceil$. Note that the class contains no graph for which this value is 7 or 8.

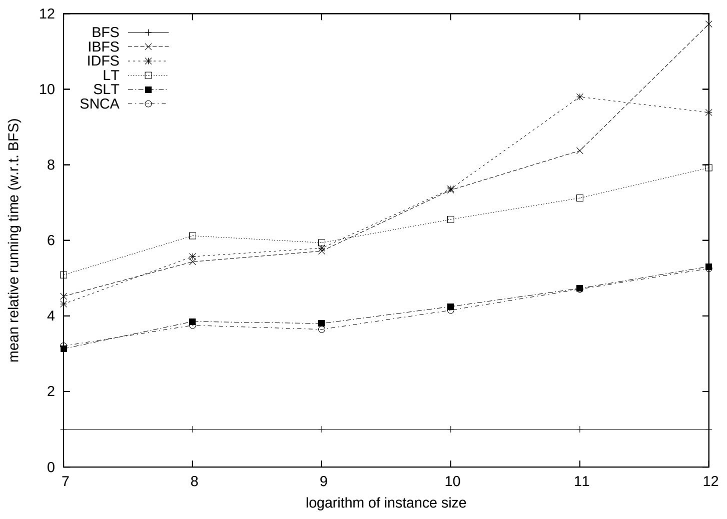

Figure 6: Times for FOODWEB instances within each size. Each point is the mean relative running time (w.r.t. BFS) for instances with the same value of $\lceil \log_2(n+m) \rceil$.

Table 3: Percentage of vertices that have their parents as semidominators (SDP), average number of iterations, and number of vertex comparisons per arc.

	SDP	ITERATIONS		COMPARISONS PER ARC				
CLASS	(%)	IBFS	IDFS	IBFS	IDFS	LT	SLT	SNCA
CIRCUITS	76.7	3.2000	2.8000	25.3	20.9	7.5	6.2	5.7
FOODWEB	30.9	2.1429	2.1905	12.1	13.0	4.9	4.3	4.3
IMPACT	73.4	1.4385	2.0686	11.1	12.2	6.2	5.1	4.4
IMPACTP	88.6	1.5376	2.0819	12.8	12.0	6.0	4.7	4.3
IMPACTS	73.4	1.4385	2.0686	11.4	12.1	6.8	5.4	4.6
SUIF-FP	67.7	1.6817	2.0000	11.9	9.2	7.5	5.9	5.1
SUIF-INT	63.9	1.6659	2.0009	11.9	10.3	7.6	5.8	5.0

Because of path compression, Lengauer-Tarjan and SEMI-NCA can handle repeated arcs in constant time.

So far, we have only compared the algorithms in terms of running times. These can vary significantly depending on the architecture or even the compiler that is used. For a more complete understanding of the relative performance of the algorithms, Table 3 shows three architecture-independent pieces of information. The first is SDP, the percentage of vertices (excluding the root) whose semidominators are their parents in the DFS tree. These vertices are not inserted into buckets, so large percentages are better for LT and SLT. On average, far more than half of the vertices have this property. In practice, avoiding unnecessary bucket insertions makes the algorithm roughly 5% faster.

The next two columns show the average number of iterations performed by IDFS and IBFS. It is very close to 2 for IDFS: almost always the second iteration just confirms that the candidate dominators found in the first are indeed correct. This is expected for control-flow graphs, which are usually reducible in practice. On most classes the average is smaller than 2 for IBFS, indicating that the BFS and dominator trees often coincide. Note that the number of iterations for IMPACTP is slightly higher than for IMPACT, since the reverse of a reducible graph may be irreducible. The small average number of iterations helps explain why iterative algorithms are competitive. In each iteration, they perform one pass over the arcs. In contrast, the other three algorithms perform a single pass over the arcs (to compute semidominators) and one (for SNCA) or two (for SLT and LT) over the vertices.

The last five columns of Table 3 show how many comparisons between vertices are performed (normalized by the total number of arcs); the results do not include the initial DFS or BFS. The number of comparisons is always proportional to the total running time; what varies is the constant of proportionality, which is much smaller for simpler methods than for elaborate ones. Iterative methods need many more comparisons; they are competitive mainly because of smaller constants. In particular, they need to maintain only three n-sized arrays, as opposed to six or more for the other methods. (Two of these arrays

Table 4: Individual graphs (times for BFS in microseconds, all others relative to BFS). The best result in each row is marked in bold.

INSTANCE			BFS	RELATIVE RUNNING TIMES				
NAME	VERTICES	ARCS	TIME	IBFS	IDFS	LT	SLT	SNCA
idfsquad	1501	2500	31	17.8	2599.0	7.8	**4.0**	9.7
ibfsquad	5004	10003	82	11015.6	10.5	9.4	5.1	**4.8**
itworst	401	10501	32	6510.0	7583.8	9.5	4.8	**4.7**
sltworst	32768	65534	2809	287.5	288.4	**8.0**	11.5	10.6
sncaworst	10000	14999	226	191.6	391.9	9.2	**4.6**	287.4
atis	4950	515080	2607	12.7	8.5	6.2	3.3	**3.2**
nab	406555	939984	47619	15.9	18.2	12.7	12.0	**10.2**
pw	330762	823330	42500	15.1	18.4	13.4	12.4	**10.6**

translate vertex numbers into DFS or BFS labels and vice-versa.)

Note that the constants associated with our implementation of IBFS are slightly smaller than those of IDFS. On SUIF graphs, for example, IBFS performs more comparisons per arc than IDFS: because IBFS starts from a full tree, it must process all arcs during the first iteration. Even so, IBFS is still faster on average, as shown in Table 2. In fact, we have observed that the initial DFS is around 40% slower than the initial BFS. This happens because, while the array that maps BFS labels into vertex numbers doubles as the queue used by BFS, DFS actually needs an extra stack. This stack can be implemented either as a separate array or (implicitly) by making the function recursive. We used the latter approach, but their performance is similar.

Table 3 also shows that SLT performs fewer comparisons per arc than LT. This happens despite the fact that the paths traversed by LT during the *eval* operation have on average fewer vertices than those traversed by SLT, as one would expect. For these graphs, the difference is clearly not large enough to offset the extra cost of maintaining a more complicated data structure.

We end our experimental analysis with results on artificial graphs. For each graph, Table 4 shows the numbers of vertices and arcs, the time for BFS (in microseconds), and the times for computing dominators (as multiples of BFS). The first five entries represent the worst-case families described in Section 3. In all cases, the algorithms behave as predicted. The speech-recognition graphs (atis, nab, and pw) have no special adversarial structure, but are significantly larger than other graphs. As previously observed, the performance of iterative methods tends to degrade more noticeably with size. SNCA and SLT remain the fastest methods, but the asymptotically better behavior of LT starts to show.

5 Final Remarks

We compared five algorithms for computing dominators. Results on three classes of application graphs (program flow, VLSI circuits, and foodwebs) indicate that they all have similar overall performance in practice. The tree-based iterative algorithms proposed by Cooper et al. are by far the easiest to code and use less memory than the other methods, which makes them perform particularly well on small, simple graphs. For the compiler-generated graphs we tested, the iterative algorithms remained competitive even as the size increased. Given their simplicity, they are a good choice for non-critical applications.

Even on small instances, however, we did not observe the clear superiority of the original tree-based algorithm (IDFS) reported by Cooper et al., which we attribute to their inefficient implementation of SLT [10]. Both versions of the Lengauer-Tarjan algorithm (LT and SLT) and the hybrid algorithm (SNCA) are more robust on application graphs, and the advantage increases with graph size or graph complexity. Among these three, LT was the slowest, in contrast with the results reported by Lengauer and Tarjan [30]. SLT and SNCA were the most consistently fast algorithms in practice; since the former is less sensitive to pathological instances, we think it should be preferred where performance guarantees are important.

Acknowledgements

We thank Spyridon Triantafyllis and David August for providing us with the IMPACT graphs and for their collaboration on the preliminary version of this work. Matthew Bridges helped us install and use the IMPACT compiler. We thank Adam Buchsbaum for providing the SUIF and speech recognition graphs and for useful comments on an earlier draft of this paper. Stefano Allesina and Jennifer Dunne gave us the foodweb graphs. We also thank the referees for several useful comments and suggestions.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[3] F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, 1972.

[4] S. Allesina and A. Bodini. Who dominates whom in the ecosystem? Energy flow bottlenecks and cascading extinctions. *Journal of Theoretical Biology*, 230(3):351–358, 2004.

[5] S. Allesina, A. Bodini, and C. Bondavalli. Secondary extinctions in ecological networks: Bottlenecks unveiled. *Ecological Modelling*, 2005. In press.

[6] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999.

[7] M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. In *Proceedings of the 19th IEEE VLSI Test Symposium*, pages 124–130, 2001.

[8] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time pointer-machine algorithms for path-evaluation problems on trees and graphs. In preparation.

[9] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–96, 1998. Corrigendum appeared in 27(3):383-7, 2005.

[10] K. D. Cooper, 2003. Personal communication.

[11] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Available online at `http://www.cs.rice.edu/~keith/EMBED/dom.pdf`.

[12] K. D. Cooper, T. J. Harvey, and K. Kennedy. Iterative data-flow analysis, revisited. Technical Report TR04-432, Department of Computer Science, Rice University, 2004.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[14] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[15] M. J. Fischer. Efficiency of equivalence algorithms. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 153–168. Plenum Press, New York, 1972.

[16] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 434–443, 1990.

[17] A. Gal, C. W. Probst, and M. Franz. Complexity-based denial-of-service attacks on mobile code systems. Technical Report 04-09, School of Information and Computer Science, University of California, Irvine, 2004.

[18] A. Gal, C. W. Probst, and M. Franz. Average case vs. worst case: Margins of safety in system design. In *Proceedings of the New Security Paradigms Workshop*, 2005.

[19] L. Georgiadis. *Linear-Time Algorithms for Dominators and Related Problems*. PhD thesis, Princeton University, 2005.

[20] L. Georgiadis and R. E. Tarjan. Finding dominators revisited. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 862–871, 2004.

[21] L. Georgiadis and R. E. Tarjan. Dominator tree verification and vertex-disjoint paths. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 433–442, 2005.

[22] L. Georgiadis, R. F. Werneck, R. E. Tarjan, S. Triantafyllis, and D. I. August. Finding dominators in practice. In *Proceedings of the 12th Annual European Symposium on Algorithms*, volume 3221 of *Lecture Notes in Computer Science*, pages 677–688, 2004.

[23] A. V. Goldberg. Shortest path algorithms: Engineering aspects. In *Proceedings of the International Symposium on Algorithms and Computation*, volume 2223 of *Lecture Notes in Computer Science*, pages 502–513. Springer-Verlag, 2001.

[24] A. V. Goldberg. A simple shortest path algorithm with linear average time. In *Proceedings of the 9th European Symposium on Algorithms*, volume 2161 of *Lecture Notes in Computer Science*, pages 230–241. Springer-Verlag, 2001.

[25] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

[26] G. Holloway and C. Young. The flow analysis and transformation libraries of Machine SUIF. In *Proceedings of the 2nd SUIF Compiler Workshop*, 1997.

[27] The IMPACT compiler. `http://www.crhc.uiuc.edu/IMPACT`.

[28] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23:158–171, 1976.

[29] D. E. Knuth. An empirical study of FORTRAN programs. *Software Practice and Experience*, 1:105–133, 1971.

[30] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979.

[31] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science*, 15:99–117, 1994.

[32] S. S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 14. Morgan-Kaufmann Publishers, San Francisco, CA, 1997.

[33] M. Poggi de Aragão and R. F. Werneck. On the implementation of MST-based heuristics for the Steiner problem in graphs. In *Proceedings of the 4th International Workshop on Algorithm Engineering and Experiments*, volume 2409 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2002.

[34] P. W. Purdom, Jr. and E. F. Moore. Algorithm 430: Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778, 1972.

[35] L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages*, volume 3819 of *Lecture Notes in Computer Science*, pages 73–87. Springer-Verlag, 2006.

[36] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3):141–153, 1980.

[37] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[38] The Standard Performance Evaluation Corp. `http://www.spec.org/`.

[39] P. H. Sweany and S. J. Beaty. Dominator-path scheduling: A global scheduling method. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 260–263, 1992.

[40] R. E. Tarjan. Testing flow graph reducibility. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pages 96–107, 1973.

[41] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[42] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):212–225, 1975.

[43] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[44] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–81, 1984.

[45] The CAD Benchmarking Lab, North Carolina State University. ISCAS'89 benchmark information. `http://www.cbl.ncsu.edu/www/CBL_Docs/iscas89.html`.